PHYS 30762: Final Project

Jonathan Francis

10083527

School of Physics and Astronomy

The University of Manchester

January 2021

Abstract

The main objectives of this project were to create an AC circuit with resistors, capacitors, and inductors, and to calculate the total impedance and phase shift of this circuit and each component. This was achieved using a polymorphic components class and a circuit class. The programme identifies each segment in the circuit and uses these segments to recursively calculate the total impedance. In addition to the minimum specification, the programme can calculate the impedance and phase shift for nested parallel and series circuits, identify all segments in a circuit, uses non-ideal components and wires, and can visualise the circuit and individual segments in the terminal. Additional C++ features include lambda expressions, template functions, exception handling, and smart pointers.

Contents

1 Introduction

The minimum specification defines a programme with an abstract class for components and derived classes for resistors, capacitors, and inductors. There is an additional class for the circuit, to which components can be added after initialisation. At minimum functionality, the programme allows the user to create a circuit of any length using a list of components. The programme prints out the magnitude and phase shift for each component, in addition to the total impedance and phase shift for the circuit.

2 Code Design and Implementation

2.1 Filing System

This programme contains five header files and one .cpp file. There is a header file for each of the following: the classes, the non-member functions, the complex member functions, the circuit member functions, and the component member functions. The .cpp file contains the main programme. The functions were broken up in this way to keep each file as brief as possible, while the same was not done for the classes to avoid having too many header files. It is understood that this is not best practice, however, limiting the total number of files was preferred.

2.2 Class Design

The programme contains three sets of classes: the complex numbers class (reused from assignment 4), the components classes, and the circuit class. The components classes follow a polymorphic structure in which the resistor, capacitor, and inductor classes inherit from the base class, and the non-ideal versions of these components inherit from their ideal versions. In addition to the "getters" and "setters" specified in the project formulation, each component is given a name, which is inherited by the non-ideal components from the ideal components. The impedance of each component is defined at the point of initialisation.

Instead of implementing a function to set the frequency for each component, frequency was defined as a static member variable of the components class. This means that only one copy of the frequency exists for the programme, which seemed appropriate as all components in a circuit will experience the same frequency [1]. This also meant that the user could change frequency directly by input. Additionally, frequency is declared inline. This was implemented so that the frequency did not have to be declared outside of the class, which improves readability [2]. A function is also added to set the impedance for each component, which was used to update their impedance after a change in frequency.

For the circuit class, an accompanying data structure was defined to store a "link" between components. This is defined as a pair of connected components. These links form the basis of the circuit class, as each circuit is a vector of links ("adjacency_list"). In addition, two new C++ features are used.

2.2.1 Sets

Sets are containers into which only unique elements can be inserted [3]. The vector of links with which the circuit is built from contains duplicates of most components, as the end of one link forms the start of another. As such, a set is declared within the circuit class to create a container of all components used in the circuit. This was to prevent duplicates occurring when

printing out a list of components. Despite this practical feature, the primary containers used throughout this programme are vectors as components frequently need to be accessed by index.

2.2.2 Smart Pointers

All components are stored as base class pointers. Unlike raw pointers, smart pointers use a destructor which contains a call to delete [4]. This destructor is invoked when the smart pointer exits scope, and, as a result, the pointer is deleted [4]. This prevents memory leak and ensures no manual garbage collection is required [4]. The type of smart pointer used in this programme is the "shared pointer". A shared pointer is used when the pointer has multiple owners, and is preferred in this project as components are frequently transferred between containers [4].

2.3 Circuit Class Functions

The main functions related to the functionality of the programme are included in the circuit member functions file. These are members of the circuit class as they all relate to operations on circuit segments and properties of the circuit. In addition, two global constants for the complex class were defined at the top of the file, representing the real numbers one and zero. This was implemented rather than overloading the assignment operator to restrict the functionality of the programme only to what is necessary. To summarise the output of these functions, every path within the circuit is identified, along with all components which begin and end a parallel section. These paths are then used to recursively compute the impedance of the circuit. Intermediary functions to compute the impedance of an individual segment and to identify all segments belonging to a parallel section are included to break up the code. A function to compute the total phase shift of the circuit is also included. The following subsections highlight the additional C++ features used in this file, besides the global constants mentioned earlier.

2.3.1 Switch Statements

Switch statements are used to alternate between a set of code blocks and execute the block with a "case" which matches a given expression [5]. These cases are denoted by positive integers, and a default case is typically included, for which the code block executes if no match is found [5]. In this respect, switch statements function like if-else statements. For multiple conditions, switch statements are often preferred as they improve readability over nested if-else blocks [6].

A switch statement is used in the function which computes the total impedance of a circuit to distinguish between a circuit with parallel sections in series, and a circuit with only one parallel section. In both cases, the parallel sections can be nested. A code snippet is shown in figure 1.

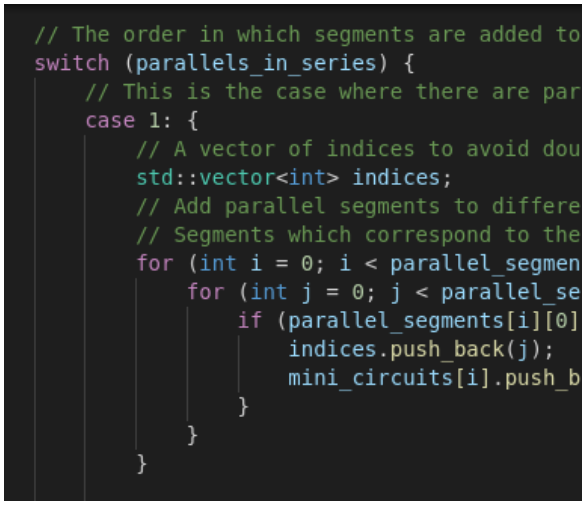


Figure 1: The beginning of the switch statement used in the "get_total_impedance" function.

As shown in figure 1, the switch statement takes "parallels_in_series" as an argument, which evaluates to 1 if there are multiple parallel sections in series for a given "layer" of the circuit. Consequently, case 1 executes if there are parallel sections in series, and the default case executes if only one parallel section is found. For very complex circuits, with multiple layers of parallel sections, the function may frequently "switch" between the two cases. For this reason, a switch statement was chosen.

2.3.2 Lambda Expressions

Lambda expressions are used to insert function-like blocks at the point where these functions would be called [7]. This is desirable when a function is required but need only be used once, for which case lambda expressions provide a more compact and efficient alternative [7].

A lambda expression is used twice in the "get_total_impedance" function for the same purpose, albeit operating on different types of containers. It is used to erase the first and last components of a segment within a parallel section (the components which start/end a parallel section) to prevent the function from calling itself indefinitely. A code snippet is shown in figure 2.

```
// This time we have to erase the first and last components of the parallel segments before adding them to the mini circuit
std::transform(parallel_segments.begin(), parallel_segments.end(), parallel_segments.begin(), [](std::vector<std::shared_ptr<component>> segment) {
    segment.erase(segment.begin());
    segment.erase(segment.begin() + segment.size() - 1);
    return segment;
});
```

Figure 2: An example of one of the lambda expressions used in this file.

As shown in figure 2, the lambda expression is called in "std::transform". This is a method which applies a function to all elements within a range [8]. The lambda expression is taken as the third argument of std::transform and has three components: a capture clause, a parameter list, and a body [7]. The capture clause allows the lambda to access variables from the surrounding scope; an empty clause in this case indicates no variables are captured [7]. The parameter list is identical to the argument of a function, and specifies the input parameters [7].

Similarly, the body is identical to the body of a normal function [7]. This lambda is applied to each element in the "parallel_segments" container.

Sets are also used in this file to eliminate duplicates from a vector. This is shown in figure 3, where each element in parallel_segments is inserted into a set and then copied back into parallel_segments using "std::copy".

```
// Transfer elements from vector to set to eliminate duplicates, then transfer back
std::set<std::vector<std::shared_ptr<component>>> temp_set;
for (int i = 0; i < parallel_segments.size(); i++) {
    temp_set.insert(parallel_segments[i]);
}
parallel_segments.clear();
parallel_segments.resize(temp_set.size());
std::copy(temp_set.begin(), temp_set.end(), parallel_segments.begin());

return parallel_segments;
```

Figure 3: A code snippet of how a set is used to eliminate duplicates in this file.

2.4 Non-Member Functions

These functions are primarily related to the main programme and are used to set wire specifications, build a container for user-defined components, add links to the circuit, set the frequency, and print components/the circuit to the terminal. The following subsections highlight the additional features used in this file that have not already been discussed.

2.4.1 Maps

Maps are containers which store a mapped value and a key value [9]. The key value is typically used as an element identifier, with the mapped value representing the information associated with the key [9]. In this programme, a map is used to hold the components defined by the user. A map was chosen so that the user could access each component in the map by name when adding them to the circuit. Components, along with their names, are added to the map using "std::pair", which stores two objects as one [10].

2.4.2 Exception Handling

C++ exceptions arise when a run-time problem occurs, such as an attempt to access an index out of the range of a container [11]. Exception handling in C++ is structured around "try-blocks" [11]. In this programme, components are created dynamically through the use of "new" upon insertion into the map. To prevent bad memory allocation, this is conducted within a try-block. An example is shown in figure 4.

```
try
{
    components.insert(std::pair(component_name, new resistor(resistance,component_name)));
}
catch (const std::bad_alloc& b) {
    std::cerr << "Memory allocation failure" << std::endl;
    throw;
}
```

Figure 4: An example of how exception handling is implemented in this programme.

The try-block contains the code for which an exception can occur [11]. The "catch-block" takes the exception it is implemented to catch as an argument [11]. In this case, "std::bad_alloc" is used to catch an exception if "new" fails to allocate the requested memory [11]. An error message is included in this catch block to tell the user what the problem was. Finally, "throw" throws the exception and stops the programme [11]. Because the objects created with new are smart pointers, they do not need to be manually deleted.

2.4.3 Function Templates

Function templates are functions which can operate on more than one type by taking a generic type as an input [12]. This way, the same block of code can be used to perform the same function for multiple types of objects. In this programme, a template is used to print out a list of components from a set and a vector. This is shown in figure 5.

```
// Template function to print a list of components with their impedances and phase shifts
template <typename DataType>
void print_components(DataType components)
{
    std::cout << "Frequency of the circuit: " << component::frequency << " Hz" << std::endl;

    for(auto it = components.begin(); it != components.end(); it++) {
        print_pointer(*it);
    }

    std::cout << std::endl;
}
```

Figure 5: A code snippet of how a function template is used to print out components in this programme.

A template parameter ("DataType") is declared before the body of the function to indicate an unspecified type for use within the function. The function is then defined in the normal sense. Within the function, a function is called to print out the properties of each element, such as impedance or name. Consequently, DataType cannot represent any type; it has to represent a type which can be used within the "print_pointer" function.

A switch statement is also used in this file to insert different components into the map of components based on user input. The non-ideal varieties are then specialised within each case. Additionally, a lambda expression is used in the "set_frequency" function to update the

impedance of each component in the components map by calling "set_impedance" for each component. This is done within "std::for_each", which behaves similarly to "std::transform".

The member functions of the derived classes are all identical, with minor differences related to impedance definitions.

3 Results and Advanced Functionality

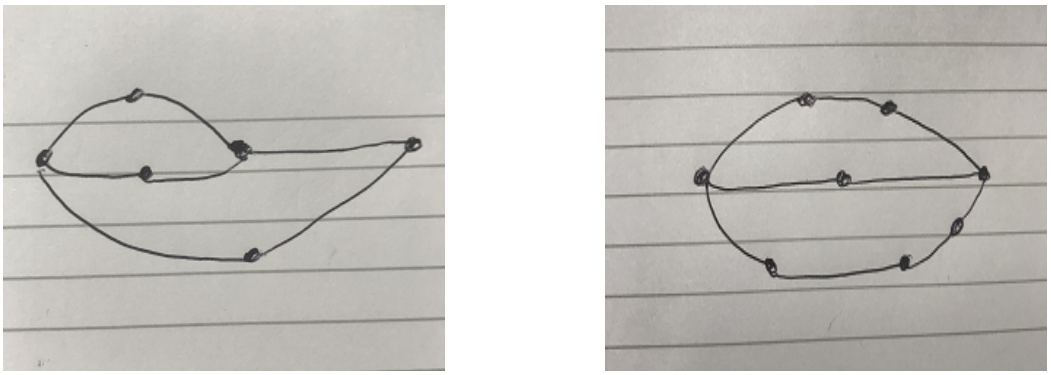
This section will briefly run through the main programme, which typically only calls functions defined elsewhere — mostly in the non-member functions file — to ease readability. Nested while loops are used so that the user can update the circuit/frequency indefinitely.

First, the user gives specifications for the wires which are used to calculate the wire contributions to inductance and capacitance. These properties are used to define the impedances of the non-ideal components. Then, the user creates the map of components, which is printed to the terminal with the name and impedance of each component. The user then adds links to the circuit by inputting the names of the starting component and the connecting component for each link. This was done to recreate the experience of assembling a circuit by hand, and give the user as much freedom as possible. Next, the user sets the frequency of the circuit. Finally, the programme prints a representation of the circuit, along with all the components used in the circuit and the frequency of the circuit. The option is then given for the user to print out the components used in each circuit segment. After each input, a while loop is used to check for bad input and prompts the user to try again if bad input is found. The following subsections highlight the additional functionality of the circuit.

3.1 Nested Parallel and Series Circuits

The programme is able to create and operate on nested parallel and serial circuits with two caveats:

- Each segment of a parallel section of a circuit must have the same start and end points. This is explained using figure 6:



(a) Not acceptable (b) Acceptable

Figure 6: A figure of two possible circuit configurations in which the points represent components and the lines represent wires.

Figure 6a demonstrates a parallel section in which each parallel segment does not share the same start and end points, which is not handled by the programme. Figure 6b, on the other hand, is acceptable.

- There must only be one component which connects back to the starting component. Consider the example shown in figure 7:

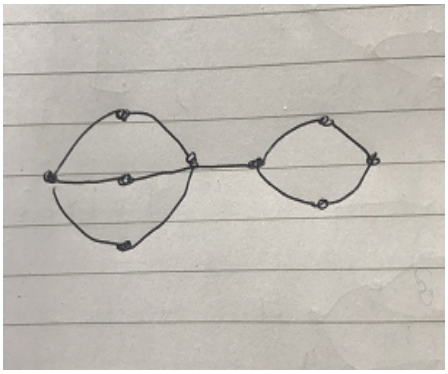


Figure 7: A figure of a possible configuration in which the points represent components and the lines represent wires.

> Let the left-hand side of figure 7 define the start of the circuit and the right-hand side of figure 7 define the end of the circuit. The right-hand side must finish on a single component, which will then implicitly loop back to the outermost left-hand component to complete the circuit.

3.2 Recursively-Calculated Impedance

The function used to calculate the impedance of the circuit is defined recursively. This way, it is theoretically possible for this programme to calculate the impedance of a circuit with an infinite number of layers. However, only a depth of three layers is claimed, as this is the limit with which the circuit has been tested.

3.3 Finding and Printing all Components in a Circuit Segment

The programme finds all possible segments in a circuit and gives the user the option to view them. This can be achieved for circuits of any complexity, and is not hindered by the aforementioned caveats. The names, phase shifts, and impedances of components are printed to the terminal, along with a visualisation of each segment. This is the preferred visualisation method.

3.4 Non-Ideal Components and Wires

The non-ideal components are distinguished from their perfect counterparts through the inclusion of parasitic elements and contributions from the wires in their impedances. In addition, they generally have impedances with both real and imaginary components.

3.5 Circuit Visualisation

Circuit visualisation is implemented in the terminal, as shown in figure 8.

```
Printing circuit (the final component connects back to the first):
res1 —> res2  res2 —> res3  res3 —> res4  res4 —> res5  res5 —> res6
```

Figure 8: An example of the circuit printing output for a series circuit with six components.

The output in figure 8 is achieved by printing each link within the vector of links. For parallel circuits, a line break divides parallel sections to ease readability.

4 Conclusions

The programme could be improved by the ability to calculate the impedance of a circuit of any complexity, such as the configuration presented in figure 6a. This is currently not supported by the "get_parallel_segments" function, which would have to be redesigned. Circuit visualisation could also be improved using the Boost Graph Library, which supports the visualisation of directed graphs (like the circuit designed in this programme). Additionally, other common circuit elements could be added, such as a bulb or a voltage source. The programme could then be adapted to simulate voltage and current and could, for example, simulate a light bulb turning on if the power is high enough.

Overleaf Word Count: 2484

References

[1] *Static member variables*, LearnCpp, accessed 17-5-2021. [Online]. Available: https://www.learncpp.com/cpp-tutorial/static-member-variables/.

[2] *Global constants and inline variables*, LearnCpp, accessed 17-5-2021. [Online]. Available: https://www.learncpp.com/cpp-tutorial/global-constants-and-inline-variables/.

[3] *Set*, CPlusPlus, accessed 17-5-2021. [Online]. Available: http://www.cplusplus.com/reference/set/set/.

[4] *Smart pointers (Modern C++)*, Microsoft, accessed 17-5-2021. [Online]. Available: https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=msvc-160.

[5] *C++ Switch*, W^3 Schools, accessed 17-5-2021. [Online]. Available: https://www.w3schools.com/cpp/cpp_switch.asp.

[6] *Why switch is better than if-else*, Musing Mortoray, accessed 17-5-2021. [Online]. Available: https://mortoray.com/2019/06/29/why-switch-is-better-than-if-else/.

[7] *Lambda expressions in C++*, Microsoft, accessed 17-5-2021. [Online]. Available: https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=msvc-160.

[8] *The difference between for_each and transform*, Angelika Langer - Training Consulting, accessed 17-5-2021. [Online]. Available: http://www.angelikalanger.com/Articles/Cuj/03.ForeachTransform/ForEachTransform.html.

[9] *Map*, CPlusPlus, accessed 17-5-2021. [Online]. Available: https://www.cplusplus.com/reference/map/map/.

[10] *Pair*, cppreference, accessed 17-5-2021. [Online]. Available: https://en.cppreference.com/w/cpp/utility/pair.

[11] *C++ Exception Handling*, Tutorials Point, accessed 17-5-2021. [Online]. Available: https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.html.

[12] *Templates*, CPlusPlus, accessed 17-5-2021. [Online]. Available: https://www.cplusplus.com/doc/oldtutorial/templates/.